The Journal of Open Source Software

t8code - modular adaptive mesh refinement in the exascale era

Johannes Holke[1*¶], Johannes Markert[1*], David Knapp[1*], Lukas Dreyer[1], Sandro Elsweijer[1], Niklas Böing[1], Chiara Hergl[1], Prasanna Ponnusamy[1], and Achim Basermann[1]

1 German Aerospace Center (DLR), Institute for Software Technology, Cologne, Germany ¶ Corresponding author * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

In this note we present our software library t8code for scalable dynamic adaptive mesh refinement (AMR) officially released in 2022 (Holke et al., 2022). t8code is written in C/C++, open source, and readily available at www.dlr-amr.github.io/t8code. The library provides fast and memory efficient parallel algorithms for dynamic AMR to handle tasks such as mesh adaptation, load-balancing, ghost computation, feature search and more. t8code can manage meshes with over one trillion mesh elements (Holke et al., 2021) and scales up to one million parallel processes (Holke, 2018). It is intended to be used as mesh management backend in scientific and engineering simulation codes paving the way towards high-performance applications of the upcoming exascale era.

Introduction

AMR has been established as a successful approach for scientific and engineering simulations over the past decades (Babuvška & Rheinboldt, 1978; Bangerth et al., 2007; Dörfler, 1996; Teunissen & Keppens, 2019). By modifying the mesh resolution locally according to problem specific indicators, the computational power is efficiently concentrated where needed and the overall memory usage is reduced by orders of magnitude. However, managing adaptive meshes and associated data is a very challenging task, especially for parallel codes. Implementing fast and scalable AMR routines generally leads to a large development overhead motivating the need for external mesh management libraries like t8code.

t8code is written in C/C++, open source, and the latest release can be obtained at https://dlr-amr.github.io/t8code (Holke et al., 2022). It uses efficient space-filling curves (SFC) to manage the data in structured refinement trees. While in the past being successfully applied to quadrilateral and hexahedral meshes (Burstedde et al., 2011; Weinzierl, 2019), t8code extends these SFC techniques in a modular fashion, such that arbitrary element shapes are supported. We achieve this modularity through a novel decoupling approach that separates high-level (mesh global) algorithms from low-level (element local) implementations. All high-level algorithms can then be applied to different implementations of element shapes and refinement patterns. A mix of different element shapes in the same mesh is also supported.

Currently, t8code provides implementations of Morton type SFCs with $1:2^d$ refinement for vertices ($d = 0$), lines ($d = 1$), quadrilaterals, triangles ($d = 2$), hexahedra, tetrahedra, prisms, and pyramids ($d = 3$). The latter having a $1:10$ refinement rule with tetrahedra emerging as child elements (Knapp, 2020). Additionally, implementation of other refinement patterns and SFCs is possible according to the specific requirements of the application.

The Journal of Open Source Software

In this article we provide a brief overview and a first point of entrance for interested developers working on codes storing data on (distributed) meshes. For further information beyond this short note and also for code examples, we refer to our Documentation and Wiki (Holke et al., 2022) and our other technical papers on t8code (Becker, 2021; Burstedde & Holke, 2016, 2017; Dreyer, 2021; Elsweijer, 2021; Holke, 2018; Holke et al., 2021; Knapp, 2020; Lilikakis, 2022).

Fundamental Concepts

t8code is based on the concept of tree-based adaptive mesh refinement. Starting point is an unstructured input mesh, which we call coarse mesh that describes the geometry of the computational domain. The coarse mesh elements are refined recursively in a structured pattern, resulting in refinement trees of which we store only minimal information of the finest elements (the leafs of the tree). We call this resulting fine mesh the forest.

By enumerating the children in the refinement pattern we obtain a space-filling curve logic. Via these SFCs, all elements in a refinement tree are assigned an index and are stored in linear order of these indices. Information such as coordinates or element neighbors do not need to be stored explicitly, but can be recovered from the index and the appropriate information of the coarse elements. The less elements the input mesh has, the more memory and runtime are saved through the SFC logic. t8code supports distributed coarse meshes of arbitrary size and complexity, which we tested for up to 370 million input elements (Burstedde & Holke, 2017).

The forest mesh is distributed, that is, at any time, each parallel process only stores a unique portion of the forest mesh, the boundaries of which are calculated from the SFC indices; see Figure 1.

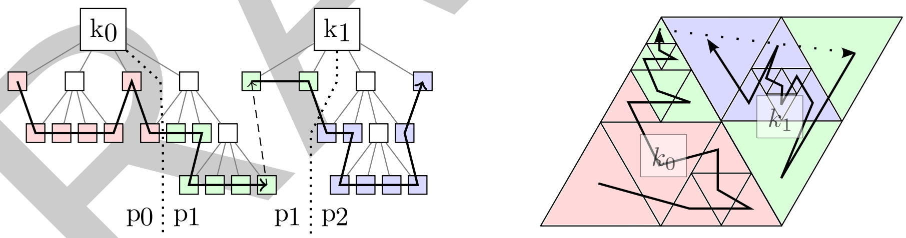

Figure 1: Left: Quad-tree of an exemplary forest mesh consisting of two trees (k_0, k_1) distributed over three parallel processes P0 to P2. The SFC is represented by a black curve tracing only the finest elements (leaf nodes) of each tree. Right: Sketch of the associated triangular mesh refined up to level three.

Interfacing with t8code

The application developer implements custom callback functions and registers them via the t8code application programming interface (API). We call this the Hollywood principle: "Don't call us, we'll call you!". Whenever an application needs to interact with the mesh, e.g., adapting the mesh, interpolating data, etc., we offer suitable callback handlers. Any mesh specific details on how to access individual elements in the forest is opaque to the application and is internally handled by t8code in an efficient manner.

In accordance to our core philosophy, any application specific data is only loosely coupled with t8code's data structures. In order to access the user data in the callbacks, the data is provided as a consecutive array with one entry per element enumerated in SFC order. For parallel applications, access to neighboring elements across parallel zones (ghost layer) is provided in a similar fashion.

The standard way to to interact with t8code is to use the high-level t8code operations: New, Adapt, Balance, Interpolate, Partition, Ghost, Iterate. This is also illustrated in the flowchart in Figure 2.

New Construct a new, uniformly refined mesh from a coarse geometry mesh. This mesh is already distributed across the parallel processes. This step is usually only carried out once during the preprocessing phase.

Adapt Decide for each element whether to refine, coarsen, or pass according to the results of a criterion provided by a custom adaption callback.

Balance Establishesa 2:1 balance condition, meaning that afterwards the refinement levels of neighboring elements are either the same or differ by at most ± 1. Note, this operation only refines elements, never coarsens them. Applications are free to decide whether they require the balance condition or not.

Interpolate Interpolate data from one forest mesh to another. For each element that was refined, coarsened or remained the same, an application provided callback is executed deciding how to map the data onto the new mesh.

Partition Re-partition the mesh across all parallel processes, such that each process has the same computational load (e.g. element count). Due to the SFC logic, this operation is very efficient and may be carried out in each time step.

Partition Data Redistribute any user defined data from the original mesh to the re-partitioned one. Input is an array with one entry for each element of the original forest containing the application data, output is an array with one entry for each element of the re-partitioned forest, containing the same data (that may previously have been on a different process).

Ghost Compute a list of all ghost elements of the current process. Ghosts are elements that are neighbors to elements of the process, but do not belong to the process itself.

Ghost exchange Transfer application specific data across all ghost elements. Input is an array of application data with one filled entry for each local element and one unfilled entry for each ghost. On output the entries at the ghost elements will be filled with the corresponding values from the neighbor processes.

Iterate Iterates through the mesh, providing face neighbor information for each element passed as an argument to the callback. In our example application, it is used to carry out the advection step of the bubble.

Search May be used additionally for extra tasks, such as searching for particles, or identifying flow features. It hierarchically iterates through the mesh and executes a callback function on all elements that match a given criterion. Leveraging the SFC tree logic, `Search` omits large chunks of the mesh if they do not match the criterion. Hence, it does not necessarily inspect each individual element and therefore performs much faster than a linear search (Burstedde, 2020; Holke et al., 2021).

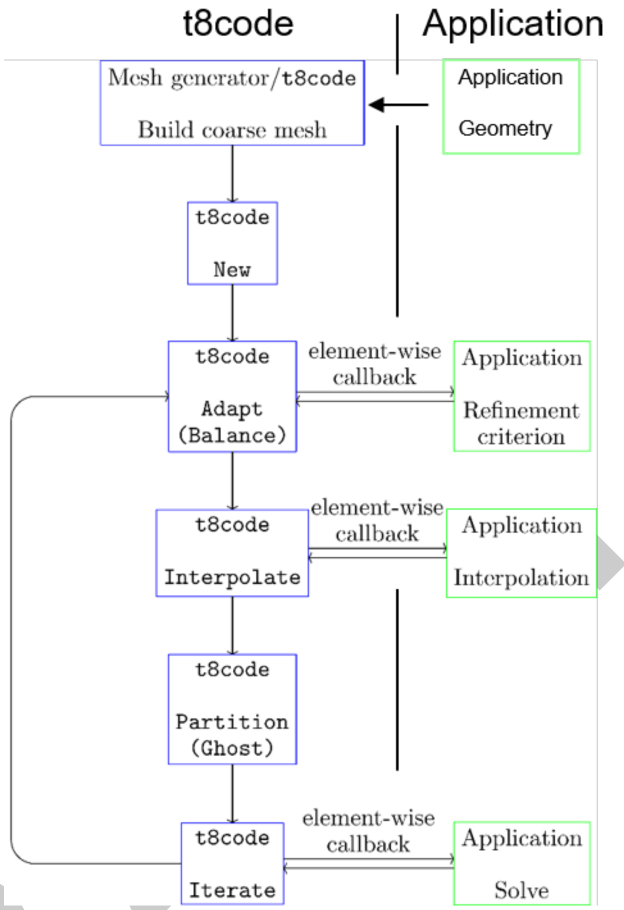

Figure 2: Flowchart of a typical simulation code which interacts with t8code. Information about the different operations can be found in the text.

Modularity & Extensibility

A distinct feature of t8code compared to similar AMR libraries is its high modularity achieved by decoupling high-level from low-level algorithms and coming along with it the support for arbitrary element shapes and refinement patterns. It also allows to combine different element shapes within the same mesh (hybrid meshes).

Thus, for each individual tree we can simply replace the underlying implementation of the low-level algorithms (e.g. from tetrahedra to hexahedra) without affecting the high-level functionality. We achieve this by encapsulating all shape-specific element operations such as parent/child computation, face-neighbor computation, SFC index computation and more in an abstract C++ base class. The different element shapes and refinement patterns are then specializations of this base class. Hence, t8code can be easily extended - also by application developers - to support other refinement patterns and SFCs.

Moreover, this very high degree of modularity allows t8code to support a wide range of non-standard capabilities. For example, the insertion of sub-elements to resolve hanging nodes (Becker, 2021) in quadrilateral meshes. Each quad element that has a hanging node is subdivided into a set of several triangles eliminating the hanging node.

Furthermore, we added support for holes in the mesh by selectively deleting elements (Lilikakis, 2022). This feature can be used to incorporate additional geometry information into the mesh. Similar to marking elements as getting refined or coarsened, we can additionally mark elements as getting removed. These elements will be eliminated completely from the SFC reducing the overall memory footprint.

Addionally, we support curved hexahedra with geometry-informed AMR (Elsweijer, 2021). Thus, information such as element volumes, face areas, or positions of interpolation/quadrature points in high order meshes can be calculated exactly with respect to the actual geometry. Another use case is to start with a very coarse input mesh and geometrically refine the mesh maxing out the performance benefits of tree-based AMR.

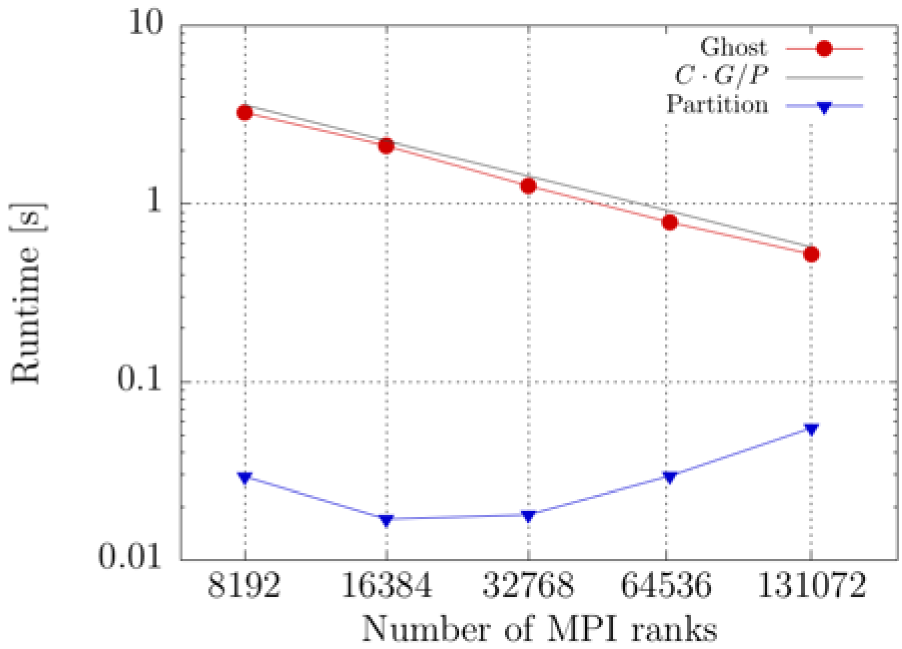

Figure 3: Strong scaling on JUWELS with tetrahedral elements. We plot the runtimes of Ghost and Partition routines with a refinement band from levels 8 to 10 after four time steps. Hence, the forest mesh consists of approximately 1.91 billion tetrahedra. As observed in the plot, we achieve perfect scaling for the Ghost algorithm in the number G/P of ghosts per process. The runtime of Partition is below 0.1 seconds even for the largest run. More details can be found in (Holke et al., 2021).

Performance

In this section we present some of our benchmark results from various performance studies conducted on the JUQUEEN (*JUQUEEN Supercomputer*, n.d.) and the JUWELS (*JUWELS Supercomputer*, n.d.) supercomputers at the Jülich Supercomputing Center. t8code's Ghost and Partition routines are exceptionally fast with proper scaling of up to 1.1 trillion mesh elements; see Table 1, (Holke et al., 2021). In Figure 3 we show a strong scaling result for a tetrahedral mesh achieving ideal strong scaling efficiency for the Ghost algorithm. Furthermore, in a prototype code (Dreyer, 2021) implementing a high-order discontinuous Galerkin method (DG) for advection-diffusion equations on dynamically adaptive hexahedral meshes we obverve a 12 times speed-up compared to non-AMR meshes with only an overall 10 to 15% runtime contribution of t8code; see autoref{fig:t8code_runtimes}.

# process	# elements	# elem. / process	Ghost	Partition
49,152	1,099,511,627,776	22,369,621	2.08 s	0.73 s
98,304	1,099,511,627,776	11,184,811	1.43 s	0.33 s

# process	# elements	# elem. / process	Ghost	Partition

Runtimes on JUQUEEN for the ghost layer and partitioning operations for a distributed mesh consisting of 1.1 trillion elements.

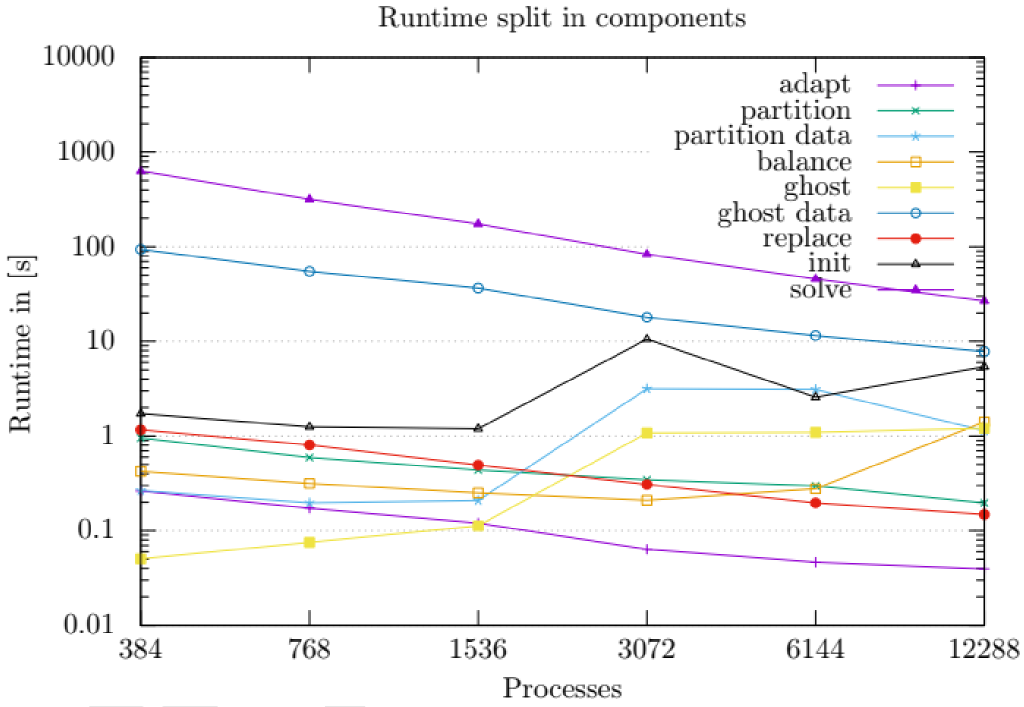

Figure 4: Runtimes on JUQUEEN of the different components of our DG prototype code coupled with t8code. Note that all features associated with dynamical mesh adaptation utilize only around 15% of the total runtime largely independent of the number of processes.

Conclusion

In this note, we introduce our open source AMR library t8code. We give a brief overview of the fundamental design principles and high-level operations. Due to the high modularity, t8code can be easily extended for a wide range of use cases. Performance results confirm that t8code is a solid choice for mesh management in high-performance applications in the upcoming exascale era.

References

Babuvška, I., & Rheinboldt, W. C. (1978). Error estimates for adaptive finite element computations. *SIAM Journal on Numerical Analysis*, *15*(4), 736–754. https://doi.org/10.1137/0715049

Bangerth, W., Hartmann, R., & Kanschat, G. (2007). Deal.II—a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, *33*(4), 24–es. https://doi.org/10.1145/1268776.1268779

Becker, F. (2021). *Removing hanging faces from tree-based adaptive meshes for numerical simulations* [Master's thesis]. Universität zu Köln.

Burstedde, C. (2020). Parallel tree algorithms for AMR and non-standard data access. *ACM Trans. Math. Softw.*, *46*(4). https://doi.org/10.1145/3401990

Burstedde, C., & Holke, J. (2016). A tetrahedral space-filling curve for nonconforming adaptive meshes. *SIAM Journal on Scientific Computing*, *38*, C471–C503. https://doi.org/10.1137/15M1040049

Burstedde, C., & Holke, J. (2017). Coarse Mesh Partitioning for Tree-Based AMR. *SIAM Journal on Scientific Computing*, *Vol. 39*, C364–C392. https://doi.org/10.1137/16M1103518

Burstedde, C., Wilcox, L. C., & Ghattas, O. (2011). p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, *33*(3), 1103–1133. https://doi.org/10.1137/100791634

Dörfler, W. (1996). A convergent adaptive algorithm for poisson's equation. *SIAM Journal on Numerical Analysis*, *33*(3), 1106–1124. https://doi.org/10.1137/0733054

Dreyer, L. (2021). *The local discontinuous galerkin method for the advection-diffusion equation on adaptive meshes* [Master's thesis, Rheinische Friedrich-Wilhems-Universität Bonn]. https://elib.dlr.de/143969/

Elsweijer, S. (2021). *Curved Domain Adaptive Mesh Refinement with Hexahedra*. Hochschule Bonn-Rhein-Sieg. https://elib.dlr.de/143537/

Holke, J. (2018). *Scalable algorithms for parallel tree-based adaptive mesh refinement with general element types* [{PhD} thesis]. Rheinische Friedrich-Wilhelms-Universität Bonn.

Holke, J., Burstedde, C., Knapp, D., Dreyer, L., Elsweijer, S., Uenlue, V., Markert, J., Lilikakis, I., & Boeing, N. (2022). *t8code* (Version 1.0.0). https://doi.org/10.5281/zenodo.7034838

Holke, J., Knapp, D., & Burstedde, C. (2021). An Optimized, Parallel Computation of the Ghost Layer for Adaptive Hybrid Forest Meshes. *SIAM Journal on Scientific Computing*, C359–C385. https://doi.org/10.1137/20M1383033

JUQUEEN supercomputer. (n.d.). FZ Jülich. Retrieved January 3, 2023, from https://hbp-hpc-platform.fz-juelich.de/?page_id=34

JUWELS supercomputer. (n.d.). FZ Jülich. Retrieved January 3, 2023, from https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/juwels

Knapp, D. (2020). *A space-filling curve for pyramidal adaptive mesh refinement* [{M}aster's Thesis]. Rheinische Friedrich-Wilhelms-Universität Bonn.

Lilikakis, I. (2022). *Algorithms for tree-based adaptive meshes with incomplete trees* [Master's thesis, Universität zu Köln]. https://elib.dlr.de/191968/

Teunissen, J., & Keppens, R. (2019). A geometric multigrid library for quadtree/octree AMR grids coupled to MPI-AMRVAC. *Computer Physics Communications*, *245*, 106866. https://doi.org/https://doi.org/10.1016/j.cpc.2019.106866

Weinzierl, T. (2019). The Peano Software-Parallel, Automaton-based, Dynamically Adaptive Grid Traversals. *ACM Transactions on Mathematical Software*, *45*(2), 1–41. https://doi.org/10.1145/3319797